新人技術者のための

ロジカル・シンキング入門 第4回

直したバグがゾンビのごとく復活する

冴木 元

システムの記事　ビギナーズ



ここではソース・コード管理のポイントについて述べる．複数のエンジニアが共同でソース（ソフトウェア・プログラムやHDLコードなど）を開発していると，誤って正しいファイルが上書きされてしまったり，適切に修正が反映されていないファイルがリリースされる場合がある．そのため，チームによる開発ではソース管理ツールの導入が不可欠である．ただし，ソース管理ツールを導入しても改善されない問題もある．本稿ではツールで対応する問題と運用ルールで対応する問題について考える．（編集部）

Fさんのプロジェクトはいまたいへん忙しい時期，いわゆる「火事場」となっています．結合テスト工程でバグが次々に発覚し，そのたびに修正に追われているからです．昨晩もあるやっかいなバグの修正にようやくけりを付け，新たにソース・コードをリリースしたばかりでした．

今日はもう一つ残った別のバグをやっつけるために，ほかのメンバとともに休日出勤したFさんでしたが，バグの切り分けに集中しようとしていたFさんのところにリリース先に出張している部下のYさんから妙な連絡が入りました．昨日リリースしたソース・コードを組み込むと，修正されたはずのバグが消えないばかりか，1週間前に修正したバグと似たような現象が再現したというのです．

「あっ，いかん．あまりテストせずにソース・コードを出したから，デグレったのか？[注]でも，早く修正版を出してくれないと自分の立場がない，とかいっていたのはYじゃないか…」．

早急につぶさなければならないバグが目の前にあって気が気ではないFさんは，念のため，修正前にはNGだったテスト項目を実行してみました．すると，何の問題もなくパスします．不審に思ったFさんは，とりあえず現状のエラーをメールにまとめて送ってくれないかとYさんに連絡しました．

30分後，Yさんから送られてきたエラーのリストを見ていたFさんの脳裏にあるでき事がよぎりました．「ちょっと待て．この間，新人のX君のソース・コード・レビューをやったばかりだな．X君のミスで何件かバグがカウントされてしまったから，注意しなかったっけ？ あの後もX君が原因を作ったバグが何件か修正されているはずだ．まさか…」．

いやな予感がしたFさんは，リリース先のYさんに頼んでソース・コードのタイム・スタンプをファイルに記録して送ってほしいと頼みました．すぐに送られてきたリストのタイム・スタンプは，一部のファイルが明らかにFさんの事務所のタイム・スタンプと一致しません．その後，リリース先と事務所のソース・ファイルを差分比較ツールで比較して判明したことは，次のような事実でした．「X君が修正した7月5日，7月10日，7月15日の修正のうち，リリース先のソース・コードに反映されているのは7月15日分のみ．7月5日，7月10日の分は修正前の記述になっている」．

読者のみなさんの中に，連日の休日出勤で疲れたFさん

注：修正版で既存機能が損なわれる事象を「デグレード（degrade）」と呼ぶ．

KeyWord　結合テスト，コード・レビュー，差分比較ツール，タイム・スタンプ，コンフリクト，ソース管理ツール，ソース・サーバ，階層化，分業化，ソフトウェア品質管理

の代わりにこのプロジェクトのソース管理を立て直してくれる人はいるでしょうか？

● ミスを確実に防ぐしくみを作る

時々，ニュースなどで医療事故の話を聞くことがあります．例えば，患者を取り違えて手術しそうになったとか，酸素と麻酔のチューブをまちがえてつないでしまったとかいったニュースです．そういうニュースを聞くと筆者は，ソフトウェア開発のしごとでありがちなソース管理のミスと重ねて考えてしまいます．

もちろん，人の命に直接かかわるしごととそうでないしごとの違いがあり，そこを無視した議論というのはある意味で乱暴に聞こえるかもしれません．しかし筆者としては，「単純作業におけるケアレス・ミス」という問題の管理の難しさが共通であるように思えてなりません．ニュースで医療事故が報道されると，よく聞かれるコメントが，「考えられないミスだ」というものです(**図1**)．でも，人間は実のところ，単純な作業を行う際に，この「考えられないミス」をしばしば起こしてしまうものなのです．

冒頭の例に挙げたFさんのプロジェクトのソース管理ミスについても，このことは言えるでしょう．こんな単純なミスは，ソフトウェア開発に携わったことがない人からみると「アホか？」と思うくらいまぬけで，不誠実な行いに思えるかもしれません．しかし，ソフトウェア開発に実際に従事したことのある人なら，大なり小なり似たような経験をしたり，あるいは耳にしたりしたことがあるはずです．ですから，このような管理ミスに対する対応策として，個人の注意力不足を問題にしたり，あるいは「お客様の製品を扱っているという自覚の徹底」といったお題目を掲げてみたりしても，恐らく根本的な解決にはなりません．ありがちな単純ミスを確実に防ぐしくみ作りを試みない限りは….

● 個人作業の場合の管理ミスの原因を探る

例によって，まず流れで作業内容をとらえ，ソース・コードの管理ミスというものがどこで発生するかを見ていくことにしましょう．手始めに，コードの改造の最初から最後までひとりだけで行う，ほかの人が介在しないシンプルな例から考えてみることにします．

図2にあるように，ソースの改造というのは，要するに，

取ってくる→改造する→提出する

というものです．だからまず，この流れのどこでミスが入るかを考えることにしましょう．

1)「取ってくる」工程の管理ミス

このときにありがちな管理ミスは，ソース・コードを集めて管理しておく場所がそもそも定まっていない，というものです．管理場所がきちんと定まっていなければ，各人がバラバラにソース・コードを更新してしまうことになります．

例えば冒頭の例で，Fさんがソース修正を終えた新人に向かって，「じゃあ，サーバのXXXというディレクトリに置いておくように」と告げてほかのメンバに周知しなかったら，次の修正からなんらかの管理ミスが発生しえます．ほかのメンバは，最新版のソース・コードがどこにあるのかを知らないからです．

また，管理場所が決まっていたとしても，「たぶん同じはずだから」といって自分が作業しているローカル・マシンにあるソース・コードに改造を加え始めてしまった場合，

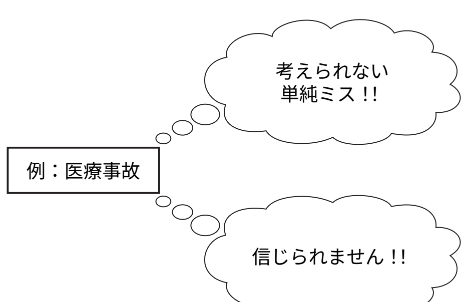


図1　単純ミスに対する反応

見た目は単純ミスであっても，再発防止に取り組むのはとても難しい．人間は機械ではないので，ほうっておけば単純ミスを起こしうる．それらを防ぐしくみ作りを考えないで，精神論のお説教で終わらせてしまうと，ミスを繰り返すはめになる

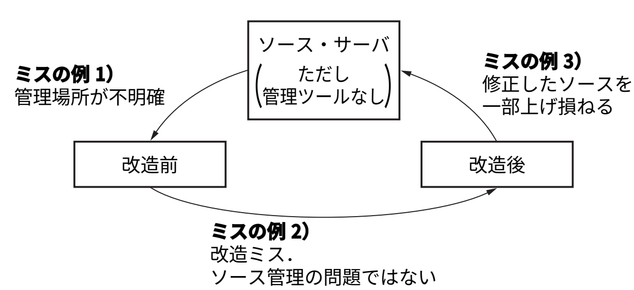


図2　ミスの発生例(個人作業の場合)

ミスの発生要因を作業全体の流れからとらえて調べてみる．これはひとりの人間のみがソース・コードに改造を加えるもっともシンプルな例．取ってくる→改造する→提出する，という流れの中で，どこでどのようなミスが生じえるのかを考える

反映されているべきだったほかの改造が抜け落ちることになりかねません．

2)「改造する」工程の管理ミス

次にミスが起こりえる場所は，改造の過程そのものです．しかしここで起こるミスは，管理ミスというより，改造作業そのもののミスです．ソース管理の問題とは別次元の問題として，とりあえず切り離してよいでしょう．

3)「提出する」工程の管理ミス

ミスは最後の提出の段階でも起こりえます．例えば，デバッグ環境というものは，ディレクトリ構成が個人によってけっこう異なるものです．だからもし，一定の管理方針がないと，追加・修正したソース・コードを一部上げ損ねることにもなりかねません．

以上で，ソース・コードを取ってきてから提出するまでの一連の流れを追ってみました．すると，ソース管理でミスが生じうる場所は，どうやら「入り口」と「出口」に集中するようです(**図3**)．すなわち，正しく取ってきて，正しく提出することができれば，ソース管理にミスは生じえないことがわかります．

● チーム作業の場合の管理ミスの原因を探る

次に，ひとりで修正が完結する場合ではなく，複数の人が介在する場合を考えてみることにします．この場合，複数の人間が同じソース・ファイルを参照・更新するため，各人はそれぞれ正しく「取得」して「提出」しても起こりえる問題があります．それは，一部のソース管理ツール(例えば版数管理や構成管理などを行うツール)で「コンフリクト(競合)」と呼ばれる事象です．

例を**図4**に示しました．時間を追ったソース・コードの履歴を矢印で記してあります．まず，Qさんが取ってきて改造に入ります．その間にPさんが取ってきて別の改造を済ませてしまいます．そこにQさんが自分の改造を反映するのです．こうなると，Pさんが施した改造がQさんによって上書きされてしまう恐れがあります．これは，PさんとQさんが同じソース・ファイルを共通して改造対象としていた場合に起こりえます．

これを防ぐには，Pさんが扱うソース・ファイルとQさんが扱うファイルを完全に別にするという手がありそうです(極端な話，1関数1ファイルに限定するソース・コード規約を作ってしまうなど)．

しかしそうしたところで，ファイルにはどうしても共有しなければチーム開発できない部分もあります．例えば，共通ヘッダ・ファイルのようなものがそれにあたります．したがって，ファイルを複数人で共有していても混乱が生じないしくみを作らない限り，解決策にはなりません．

以上で，ひとりで改造する場合と複数人で改造する場合について，順を追ってソース管理ミスが発生しうる場面を検証してきました．ひとりで改造する場合はソース・コードを取ってくる「入り口」と提出する「出口」に問題が生じがちであること，複数人で改造する場合は「コンフリクト」という事象が生じることを述べてきました．

したがってソース管理とは，今まであげたようなミスをすべて防ぐためのしくみ作りであることが理解していただけると思います．

● 管理ツールの採用は避けられない

ここではまず，「チーム開発においてはなんらかの管理ツールの採用が不可欠である」というかなりあたりまえの結論を採用したいと思います．**図5**にチーム開発向けソース管理の全体像をまとめました．

ツールをいっさい用いずに，なんらかの管理ルールのよ

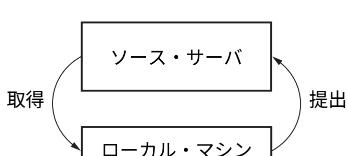


図3　ソース修正の流れ

ソース修正を流れでとらえて整理してみると，要するに「入り口」と「出口」に着目すればよさそう，ということがわかる．正しく取ってきて正しく更新すればミスは起こりえない．管理ツールも，結局はこれらを補助する道具にすぎないことに注意

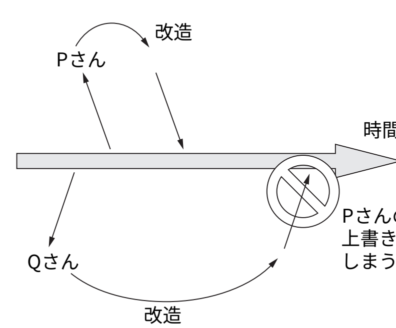


図4　ミスの発生例(チーム作業の場合)

複数の人が改造にかかわった場合のミスの発生要因を考えてみる．複数人が同じソース・ファイルを参照・更新している，という点がポイントとなって，チーム開発特有のミスが生じうる．これは「コンフリクト」と呼ばれる

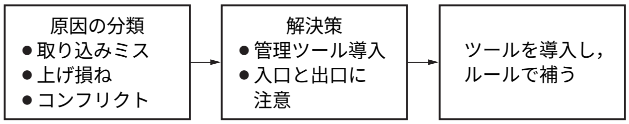


図5　チーム開発向けソース管理の全体像

まず，ミスは入り口と出口で起こりうることを整理する．コンフリクトは複数人が同じファイルにアクセスすることによって起こる．これらが管理ツールとその運用ルールによってどのように防げるかを，次に検討する

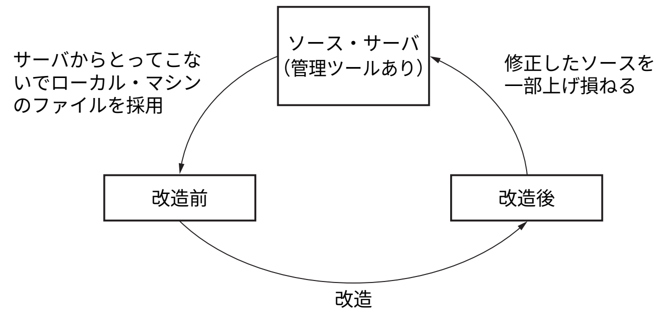


図6　管理ツールを導入しただけでは救えないミスの例

入り口と出口に着目すると，それぞれ「ソース・サーバから正しく取ってこない」，「サーバに上げ損ねた」場合は，管理ツールの外で起こるミスなのでルールが必要

うなものだけで乗り切ろうとすると，どんなにがんばってもどこかで破たんをきたします．数人で構成されるチームなら，まだなんとかなるかもしれません．しかし，複数のチームが集まって数百人にのぼる人数で作り上げるプログラムにまったく管理ツールが導入されていないとしたら，もはやそれは自殺行為です．

次のような例を想像してみましょう．計10チームで開発を行っており，各チーム合計3人ずつが1週間でソース更新を行うとします．

まず問題となるのが，各チーム内のソース管理です．チームのメンバは，各人がまちがいなく最新のソースを入手し，改造後は正しく管理元を更新しなければなりません．なおかつ，コンフリクトを防ぐための完ぺきなルール作りとその徹底が必要となります．だれがどのような順番で改造を開始し，更新しても，矛盾や上書きが生じないことが必要になります．このような事態に対して，管理ツールを導入せずになんらかの手順のルール化で対応しようとした場合，チーム内の人数が増えれば増えるほど難易度が増していきます．

さらに問題となるのはチーム間の整合性です．通常は，チームをまたがってソースを共有することはないでしょうから，コンフリクトの心配はないかもしれません．しかし，最新のソースの組み合わせがどれなのか，ということについて，各チームがまちがいなく把握できるしくみが必要です．各チーム内でもそれがおぼつかない状況で，チームをまたがってそれができるのでしょうか？

軽く想像してみただけでも途方もない話になることはおわかりいただけると思います．

● ツールの導入だけですべてを解決することはできない

問題は，「管理ツールを導入すれば，即座にすべての問題が解決するのか？」ということです．それを検討するために，管理ツールを導入すると，何が保護されるのかを考えてみることにしましょう．言い換えれば，管理ツールを導入しても，それだけでは防げないミスはどういうものかを考えることになります．これは，管理ツールで救えるミスと導入しただけでは救えないミスを切り分けることで，導入後のルール作りの基礎とするのが目的です．

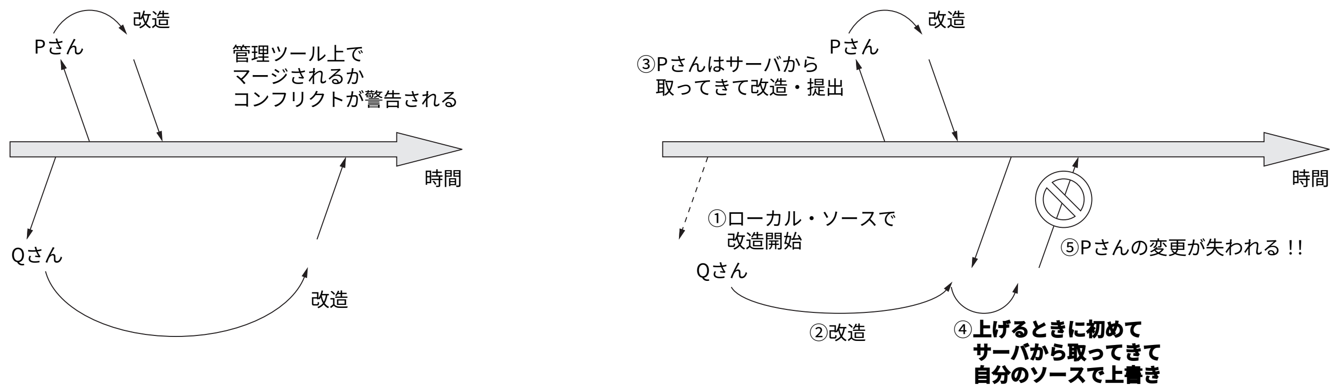
1）入り口のミス

まず，ソース・コードを取ってくる入り口のミスから考えてみます．**図2**と**図6**を比べてみてください．管理ツールが導入されていれば，ソース・サーバに最新版があるのでまちがいようがないように思われます．しかし，ソース修正に夢中になり，ローカル・マシンにあるソースの組み合わせを「最新版のつもり」で改造してしまう人が現れたらどうなるでしょう？ ソースを修正して提出したときに問題が発生しうることになりますね．なぜなら，改造する対象のソースが，ソース・サーバで管理されていたソースの最新版と異なるかもしれないからです．

したがって，まずソースを取ってくる入り口の部分では，「改造前にかならずソース・サーバからソースを取ってくること」というルールを徹底しなければならないことがわかります．管理ツールにも，変更が開始されると，ソース・サーバ上にそのことが記録されます．これがないと，管理ツールの外で改造が始まることになってしまいます．

2）出口のミス

次に，出口の部分を考えてみましょう．正しく自分のソースをソース・サーバに上げれば改造は無事完了します．しかし，管理ツールがなかったときに起こりえた「上げ損ね」，すなわち必要なファイルを過不足なく正しく上げる作業そのものは管理ツールがやってくれるわけではありません．ソースを過不足なく提出するのは，管理ツールを使



(**a**) コンフリクトを防げる例　　(**b**) コンフリクトを防げない例

図7　コンフリクトへの対応
(b)のように，サーバからソースを取得しないで改造を始めてしまい，更新するときに初めて取得して改造ソースで上書きして上げた人がいたとする．このケースは，管理ツールだけでは救えない

表1　管理ツール導入後のルール

場　所	ルール
入　口	ソース・コード改造前にソース・サーバからソースを入手し，改造を始めた記録が管理ツールに残るようにすること
出　口	ソース・コードを提出した後は，ローカル・マシンのソース・コードと一致して過不足がないかを確認すること（管理ツールの差分比較機能などを用いる）

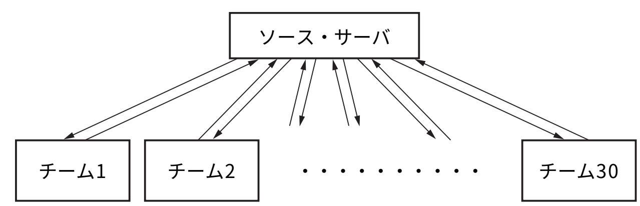


図8　30のチームからなる非常に大規模な開発の例
「ソース・コード全体が巨大で扱いづらい」，「どれが最新版かわかりにくい」という問題点が生じる．小規模の開発と同じようにソースを管理していたのでは，追いつかない

う人間の役目です．

したがって，出口の部分でも一定のルールが必要となります．すなわち，「サーバに上がったソースが手元のファイルと一致することを確認する」というルールです．一致を確認する方法としては，管理ツールに付属の差分比較ツールを用いてもよいし，何かのテキスト・エディタの機能を使ってもよいでしょう．

3) コンフリクト

次に，コンフリクトのミスを管理ツールでどのように防げるかを**図7**で考えてみることにします．**図7(a)**に示すように，正しく管理ツールを使いこなせばコンフリクトによる変更の上書きは起きません．しかし，**図7(b)**で示すような入り口のミス，すなわち正しくサーバからソースを取ってこずに改造を始めてしまうミスを犯すと，やはり変更の上書きは起こってしまいます．ソース・サーバが出どころではないソース・コードが途中から混じり込んでしまえば，やはり正しく変更が反映されなくなってしまうからです．

まとめると，管理ツール導入後は，入り口と出口に**表1**のようなルールを設けなければならないことになります．これらのルールは重要です．きわめて簡単なものであるため，遵守を徹底することは容易なはずです．

● さらに大規模な開発は階層化・分業化で対応

以上で述べてきたようなソース管理の運営は，比較的小規模な開発の場合なら，そのまま適用することが可能です．しかし，世の中で行われている商用の組み込みシステム開発の中には非常に規模の大きいものもあります．最後に，そのような開発の場合について，少しだけ考えてみることにしましょう．

図8を見てください．ここでは30のチームからなる開発を考えました．管理ツールは導入済みであったとしても，これだけ規模が大きくなると，ソース・コード全体が巨大になります．ですから，すべてまとめて更新するとなると，扱いに困ることがあります．

管理ツールがあるので，ソースの最新版はサーバ上でつねに把握されている状態を保てるでしょう．しかし，巨大なソースをまるごとソース・コードのまま扱っていると，

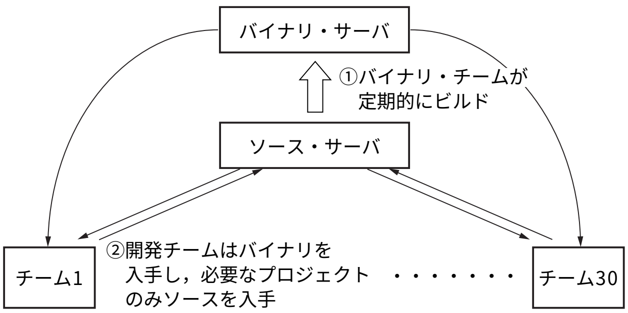


図 9　大規模開発は階層化で対応

規模が大きすぎて，全体像が見えない場合，階層化してしまうのが一つの解決策．ソース・コードをビルドしてできるバイナリ・コードとソース・コードの管理を分ける．定期的にバイナリをビルドし直す体制を整え，必要なソース・コードのみを入手して開発できるようにする

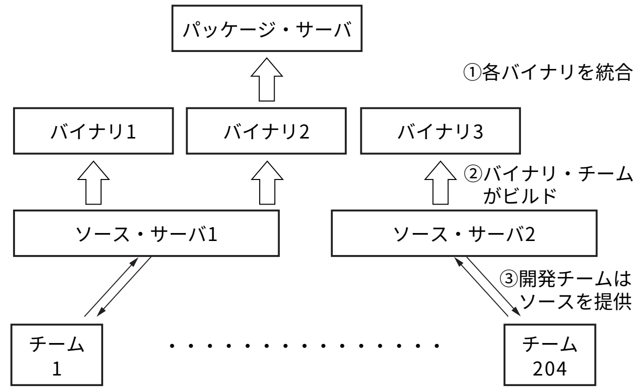
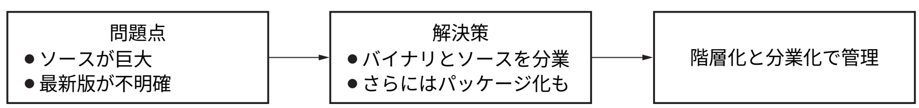


図 10　さらなる大規模開発は階層化・分業化で対応

バイナリも多様に存在して管理が容易でなくなった場合には，全体をパッケージ化してしまい，必要なバイナリとソースだけ入手すれば開発できる体制を作る．もちろん，各階層には専用の管理者が必要となる

図 11
大規模開発向けのソース管理方針の全体像

管理対象が巨大化したら，必要な範囲で階層化し，各階層に管理者を設けて対応する，というのが解決策のポイント．実際の開発でどの階層まで必要かは，個別の事情による



最新版の開発を各開発チームに強いるのはだんだん難しくなってくるはずです．なぜなら，多くのチームが頻繁に更新を繰り返すため，どのような組み合わせのソース・コードが最新版であるのか，だれにもわからなくなってしまうからです．最新版のソース・コードを確定しつつ，安定して開発を進めるためにはさらにくふうが必要となってきます．

このような場合には，**図 9** のような解決策があるでしょう．ポイントは階層化です．開発チームが開発するソース・コードの全体をビルドアップする別のチームを用意して，定期的にビルドを行うようにするのです．各開発チームはビルドのアナウンスを受けて最新のバイナリを入手し，必要なプロジェクトのみソース・コードで入手することになります．

さらに開発が大規模化した場合も，階層化・分業化で乗り切ることができるでしょう．バイナリ・コードが巨大化し，一つでは管理しきれなくなったら，機能ごとに分化して別管理とし，最後にパッケージ化する方法が考えられます(**図 10**)．各開発チームはパッケージ・バイナリ・ソースの必要なものを入手して開発することになります．まとめとして，**図 11** に大規模開発向けのソース管理方針について示します．

● ソース管理はソフトウェア品質管理の根幹

ソース管理のミスは，一見くだらないミスであるかのような印象を与えます．にもかかわらず，ソース管理はソフトウェアの品質の根幹にかかわる重要なものです．ですから，ソースを管理するということは，検討するとたいへんに奥の深いものなのです．

ソース管理の巧みなリーダは，恐らくほかの事象の管理も巧みにこなせるのではないかと，筆者は個人的に考えています．

さえき・はじめ

<筆者プロフィール>
冴木 元．システム・エンジニア．最近，ブログがはやりだが，いまだにやろうとしたことがない．知り合いにも少なからずブログをつけている者がいるのだが，なぜかつける気にならない．小学生のときにつけさせられた絵日記がよほどこたえたのか？